9.2 Defining Our Own Data Types, Part 3

All the way back in Chapter 4, we learned how to create our own simple data types in Python using the `@dataclass` decorator. While data classes are very useful, they are just one particular form of classes in Python. The `@dataclass` decorator takes our data class declaration—its *public interface*—and automatically creates an implementation of class. This makes it very simple to set up data classes, at the cost of flexibility of implementation.

In this section, we'll learn about how to create a Python data type from scratch, without the automatic implementation that `@dataclass` provides. In future sections, we'll apply what we've learned to defining new Python data types to solve various computational problems.

What if we just remove the `@dataclass`?

To start with, recall the `Person` data class example we used when we first introduced data classes:

```
@dataclass
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str
```

We were able to use this data class to create and display an instance of the class and access its attributes:

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
>>> david
Person(given_name='David', family_name='Liu', age=100, address='40 St. George
        Street')
>>> david.given_name
'David'
>>> david.family_name
'Liu'
>>> david.age
100
>>> david.address
'40 St. George Street'
```

Now let's see what happens if we remove the `@dataclass` decorator from our class definition. This is indeed valid Python syntax, but with perhaps an unexpected consequence.

```
# @dataclass (We've commented out this line)
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str
```

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
TypeError: Person() takes no arguments
```

Okay, something went wrong. Even though our class declaration still contains attribute names and type annotations, we cannot call `Person` and pass in values for those attributes. According to the error message, `Person() takes no arguments`. So what happens when we try to create an instance of `Person` and pass in zero arguments?

```
>>> david = Person()
>>> type(david)
<class 'Person'>
```

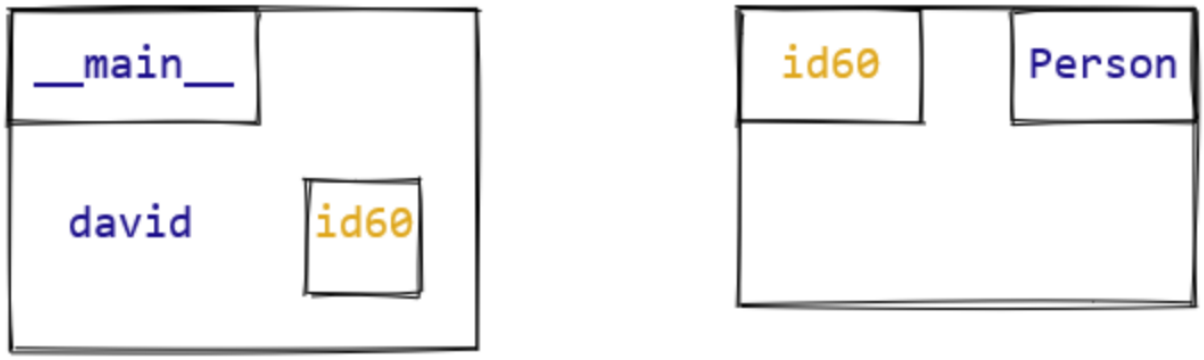
We successfully created an instance of the `Person` class. But what happens when we try to access the instance attributes?

```
>>> david.given_name
AttributeError: 'Person' object has no attribute 'given_name'
```

This should make sense: by just calling `Person()` with no arguments, we haven't specified values for any of the instance attributes, so we shouldn't expect to see a value when we access `david.given_name`.

Defining an initializer

When we execute the statement `david = Person()`, all we have in memory is this:



A `Person` object has been created, but it has no attributes. To fix this (without using `@dataclass`), we need to define a new method for `Person` called the **initializer**. The initializer method of a class is called when an instance of the class is created in Python. As its name suggests, the purpose of this method it to initialize all of the instance attributes for the new object. To distinguish it from regular functions, Python always uses the name `__init__` for the initializer method.

When we use the `@dataclass` decorator, the Python interpreter automatically creates an initializer method for the class. So let's start by seeing what this "automatic" code for the initializer looks like.

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address:
         str) -> None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
        self.address = address
```

Since all methods are functions, it should not surprise you to learn that we define methods using the same keyword (`def`) as other functions. However, there are two key differences between this method definition and all top-level function definitions we've studied so far. The first is that this method definition is *indented* so that it is inside the body of the `class Person` definition. This is how we signal that the function being defined is a method for the `Person` class.

The second difference is the presence of the parameter `self`. Every initializer has a first parameter that refers to the instance that has just been created and is to be initialized. By convention, we always call it `self`. This is such a strong Python convention that most code

checkers will complain if you don't follow it.[1] In fact, this convention is so strong that we

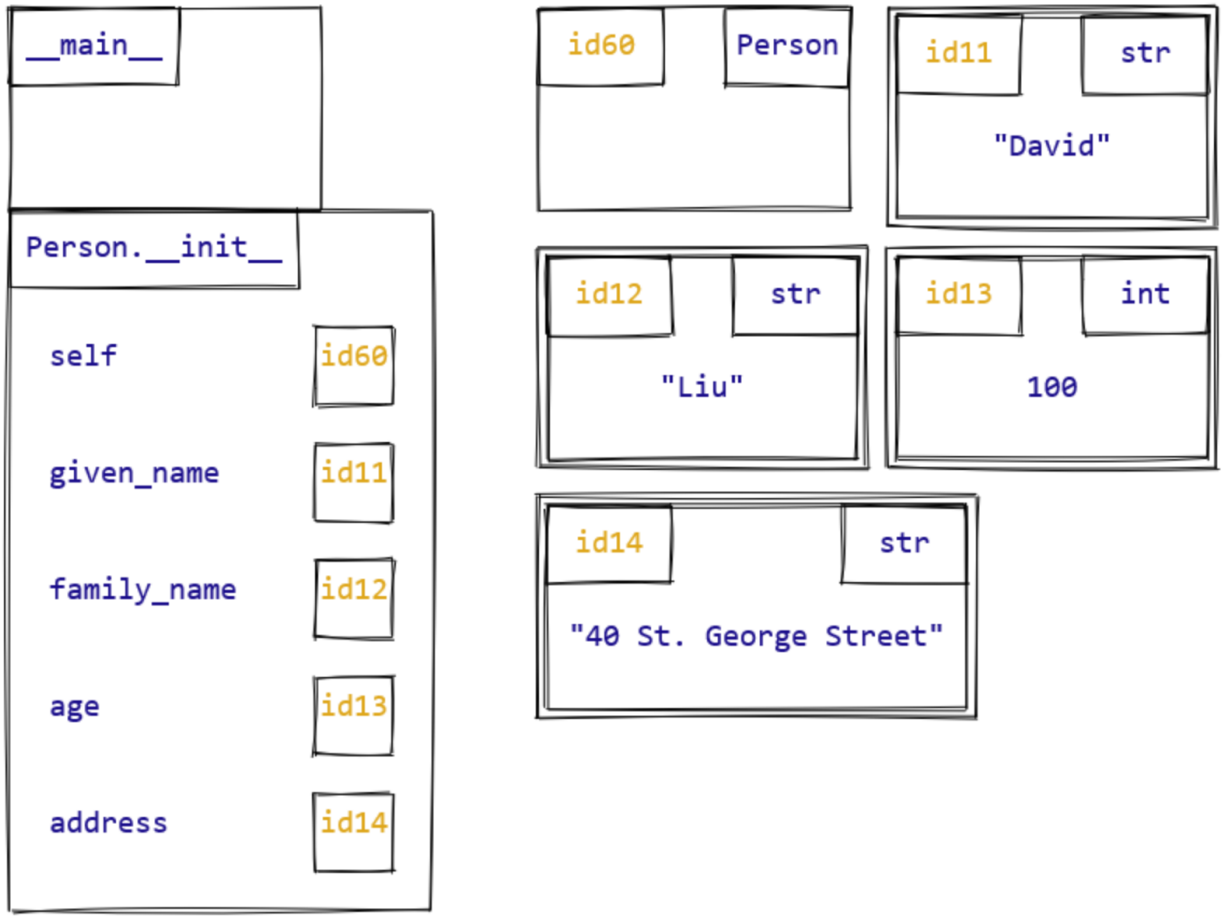
[1] This name is the reason we refer to attributes as `self.<attr>` in class representation invariants.

also typically omit the type annotation for `self`. We could have written `self: Person`, but because the type of `self` should *always* be the class that the initializer belongs to, this is considered redundant in Python!

To understand how `self` works, let's examine how we use the initializer:

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
```

Notice that we never mention the initializer `__init__` by name; it is called automatically, and the values in parentheses are passed to it. Also notice that we pass four values to the initializer, even though it has five parameters. We never have to pass a value for `self`; Python automatically sets it to the instance that is to be initialized. So this is what is happening in memory at the beginning of the initializer:

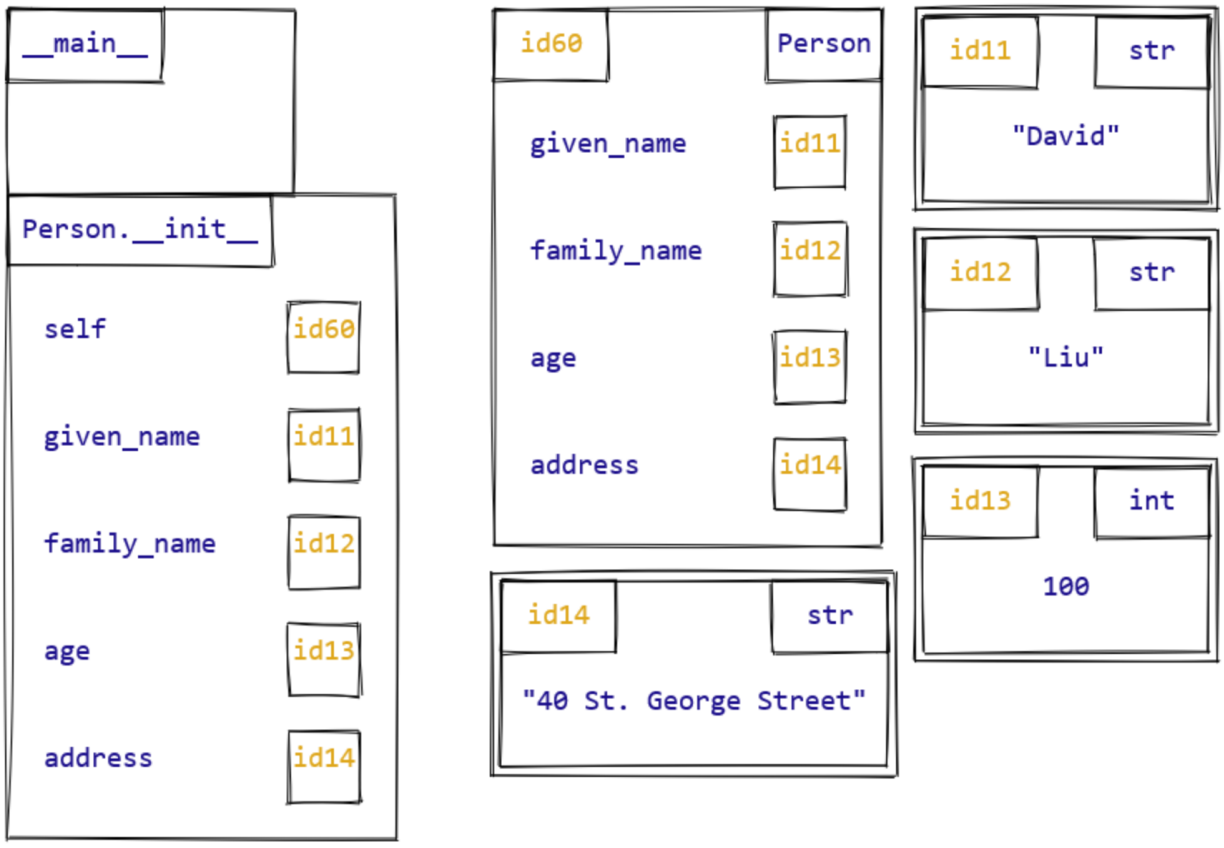


The initializer's job is to create and initialize the instance attributes. To do this, we use one assignment statement per instance attribute. This uses the same dot notation syntax that we saw in Chapter 5 for assigning to instance attributes: `self.given_name = given_name`, for example. Note that `given_name` and `self.given_name` are two different expressions!

`given_name` is a *parameter* of the initialize, while `self.given_name` is an *instance attribute*.[2] We

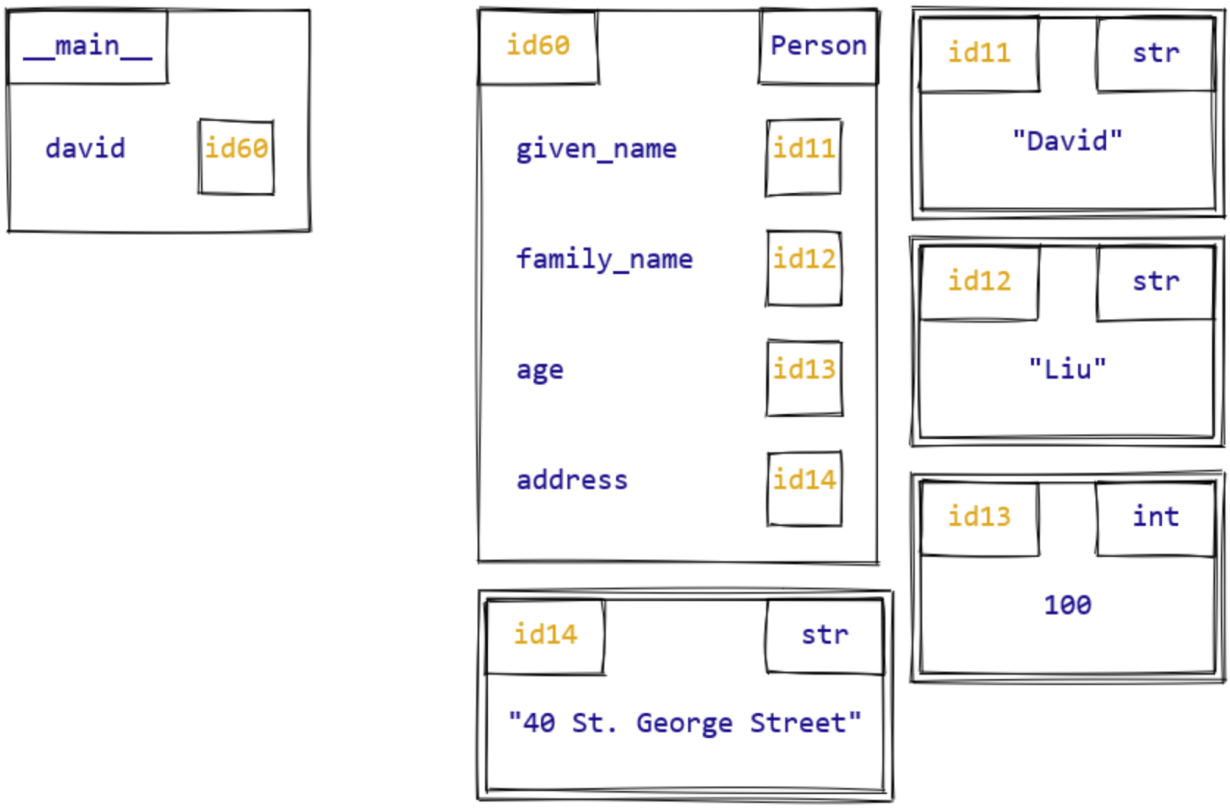
[2] Some other programming languages like Java allow you to refer to instance attributes without using dot notation. In Python, however, dot notation is *mandatory* for accessing and assigning to instance attributes.

can illustrate this distinction by showing the state of memory after all attributes have been initialized, immediately before the initializer returns:



What really happens when we create a new object

You may have noticed that the initializer return type is `None`, and that the body of the function does not actually return anything. This is a bit strange, since when we evaluate `david = Person('David', 'Liu', 100, '40 St. George Street')`, a `Person` object is definitely returned from the function call and assigned to the variable `david`.



What's going on? It turns out that calling `Person` doesn't just cause `__init__` to be called. It actually does three things:[3]

[3] Of course, this is true not just for our `Person` class, but in fact *every* class in Python.

1. Create a new `Person` object behind the scenes.
2. Call `__init__` with the new object passed to the parameter `self`, along with the other arguments.
3. Return the new object. This step is where the object is returned, not directly from the call to `__init__` in Step 2.

So in fact, `__init__` is a *helper function* in the object creation process. Its task is only to initialize attributes for an object; Python handles both creating the object beforehand, and returning the new object after `__init__` has been called.

Methods as part of a data type interface

It is certainly possible to accomplish everything that we would ever want to do with our `Person` class by writing top-level functions, and this is the approach we've taken with data classes up to this point. An alternate and commonly-used approach is to define *methods* for a data type, which become part of the interface of that data type. Remember that methods are just functions that belong to a data type—but this "belonging to" is not just an abstract relationship, but creates concrete consequences for how the Python interpreter handles them. When we define a data class and top-level functions, the interface of a data class itself only consists of its attributes; we have to remember to import those functions

separately in order to use them. When we define a class with methods, those methods are *always* bundled with the class, and so any instance of the class can use those methods, without needing to import them separately.

We have seen one example of a method definition already: the initializer, `__init__`. More generally, any function that operates on an instance of a class can be converted into a method by doing the following:

- Indent the function so that it is part of the class body, underneath the innstance attributes.
- Ensure that the first parameter of the function is an instance of the class, and name this parameter `self`.

For example, suppose we had the following function to increase a person's age:

```
def increase_age(person: Person, years: int) -> None:
    """Add the given number of years to the given person's age.

    >>> david = Person('David', 'Liu', 100, '40 St. George Street')
    >>> increase_age(david, 10)
    >>> david.age
    110
    """
    person.age = person.age + years
```

We can turn `increase_age` into a `Person` method as follows:

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address:
         str) -> None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
        self.address = address

    def increase_age(self, years: int) -> None:
        """Add the given number of years to this person's age.

        >>> david = Person('David', 'Liu', 100, '40 St. George Street')
        >>> Person.increase_age(david, 10)
        >>> david.age
```

```
        110
        """
        self.age = self.age + years
```

Notice that we now use parameter `self` (without a type annotation) to access instance attributes, just as we did in the initializer. In our function docstring, the phrase "the given person" changes to "this person",[4] and our doctest example changes the call to

[4] We typically use the word "this" in a method docstring to refer to the object instance that `self` refers to. In fact, some other programming languages also use `this` instead of `self` as a variable or keyword to refer to this object in code.

`increase_age` to `Person.increase_age`.

Shortcut syntax for method calls

Now that we are starting to define our own custom classes and methods, we are ready to see a shorthand for calling methods in Python. Let's take a look at the method call from our doctest above:

```
>>> Person.increase_age(david, 10)
```

This uses dot notation to access the `increase_age` method of the `Person` class, calling it with the two arguments `david` and `10`, which get assigned to parameters `self` and `years`, respectively.

The alternate form for calling the `increase_age` method is to use dot notation *with the `Person` instance directly*:

```
>>> david.increase_age(10)
```

When we call `david.increase_age(10)`, the Python interpreter does the following:

1. It looks up the class of `david`, which is `Person`.
2. It looks up the `increase_age` method of the `Person` class.
3. It calls `Person.increase_age` on `david` and `10`. In other words, the interpreter *automatically* passes the value to the left of the dot (in this case, the object `david` refers to) as the method's first parameter `self`.

This works not just for our custom class `Person`, but all built-in data types as well. For example, `list.append(lst, 10)` can be written as `lst.append(10)`, and `str.lower(s)` as simply `s.lower()`. More generally, a method call of the form `obj.method(x1, x2, ..., xn)` is equivalent to `type(obj).method(obj, x1, x2, ..., xn)`.

Though we've been using the more explicit "class dot notation" style (`Person.increase_age`) so far in this course, we'll switch over to the "object dot notation" style (`david.increase_age`) starting in this chapter, as this is the much more common style in Python programming. There are two primary reasons why the latter style is standard:

1. It matches other languages with an *object-oriented* style of programming, where the object being operated own is of central importance. Because we read from left to right, every time we use dot notation with the instance object on the left, we are reminded that it is an object we are working with, whether we are accessing a piece of data bundled with that object or performing an operation on that object.

   We read `david.age` as "access `david`'s age" and `david.increase_age(10)` as "increase `david`'s age by 10". In both cases, `david` is the most important object in the code expression.

2. Only the "object dot notation" style of method call supports *inheritance*, which is a technical feature of classes that we'll discuss in the next chapter.